Economics in Sharded Blockchain

Illia Polosukhin and Team
/ilblackdragon
illia@nearprotocol.com

Aug 2019

1 Introduction

Incentives are a crucial part of any decentralized protocol. These incentives must balance out to provide value to all the participants of the protocol, and ensure the protocol's future development.

Because the goal of sharded smart contract blockchains is to ultimately provide the most usable network for users and developers, a lot of thought must be put into designing the systems in place to ensure constructive behavior of all participants. Below we'll review the specific economic decisions that must be made, and the logic behind them.

In the section 2 we will review the differences between dynamic sharded Proof-of-Stake blockchains (see definition 2.1) and currently prevalent Proof-of-Work and Proof-of-Stake chains. Many specifics in the background section will refer to Nightshade ([4]).

In Proof-of-Stake platforms, there are a few different roles. These roles are not mutually exclusive (e.g. a validator can also be a developer), but a distinction is helpful due to the differing focus and goals of each party.

- Validators - provides compute, storage and security in the network in return for rewards from the protocol.
- Developers - build profitable applications, powered by underlying infrastructure of the protocol.
- Users - users of applications, and platform itself, are driven by getting value out of these interaction.
- Token Holders - holders of protocol (native) token, either for later usage or to provide liquidity.
- Protocol Governance Body - entity responsible for development and governance of the network. This can be a DAO and/or non-profit foundation.

Explicitly separating the protocol governance body, allows us to account for it's capital requirements when designing reward system (see section 7 for details).

A careful balance is required to manage the various priorities of each party. Below we will cover economics incentives for validators in section 5 and developers in section 6.

2 Background

The most studied blockchain networks are Bitcoin and Ethereum. These systems are naturally have inherent network capacity limit (transaction throughput and storage) as they function as single computer.

In contrast to these networks, sharded blockchains scale capacity linearly with number of participating nodes - each shard can accept and process transactions in parallel. Specifically, a dynamic sharded blockchain allows to maintain a network which almost never has a capacity issues by re-balancing the load based on changing circumstances.

Definition 2.1. Dynamic sharding. A dynamically sharded system is a system that can reconfigure partitioning of data and processing dynamically, without having to shut down the operating system. This enables balancing CPU / storage resources maintaining load and scale up or down depending on changing requirements.

We specifically describe details of a "Nightshade" sharding design (Skidanov and Polosukhin [4]), which allows an effective way to address the "shard of stake" issue.

Because of re-sharding, an application (or account) is not defined by the shard it is located in. This is contrasted with static sharding, where each application developer decides which shard to be deployed to. In a statically sharded system, we will observe the concentration of applications in single shard due to data and asset dependencies.

For example, lots of applications would want to be in the same shard as MakerDAO to provide users a way to interact with DAI without additional overhead. This means there will be more demand for DAI on that shard and less demand on other shards. Also there will be more demands in ETH, as users of MakerDAO will need to be able to react quickly to changes in pricing to top off there Collaterized Debt Position. This all creates very perverse incentives and goes against the idea of scaling the network.

Because of re-balancing in dynamically sharded systems, this issue is alleviated because the incentive to "co-locate" your application is absent.

To address the "shard of stake" problem, where each shard has only a small subset of total security of the system, "Nightshade" design splits selected validators into two groups: block producers (or collators) and "hidden" validators.

Block producers are responsible for receiving transactions, producing chunks, exchanging the chunks and parts of the chunks between each other while keeping

data available for other parties in the system.

"Hidden" validators are spread among all the shards and provide security for the system by keeping block producers in check, making sure they produce correct blocks and that the data is indeed available.

Because validators are hidden and do not produce blocks, e.g. there is no known assignment between validators and shards, the protocol cannot actually directly allocate rewards to validators at the moment of block production. Instead, they are rewarded for doing their work over the span of the epoch. See more details about it in section 5.2.

While sharded systems require (most likely different) pricing for transactions within a shard and also for cross-shard transactions, dynamic sharding allows all charges to be priced the same, removing the price distinction between cross-shard and intra-shard transactions.

3 Rewards

In any blockchain protocol validators (or miners for Proof-of-Work) provide their resources (compute, storage, network) in exchange for a reward. These rewards are usually a combination of coinbase (new native tokens minted) and transaction fees.

As detailed above in the "Nightshade" design, all of the logic is done on an "epoch" level, and validators get elected and rotated every N blocks. Because of this we also define rewards per epoch. At the end of every epoch, the rewards are then distributed between Validators, Developers and the Protocol Treasury.

Total epoch reward can be calculated as:

$$epochReward_t = coinbaseReward_t + epochFees_t \tag{1}$$

Where $epochFees$ is combination of all the fees collected during the blocks of the epoch t, and include transaction fees and state rent. Note, that $epochFees$ do not contain part of user paid fees that were allocated to an applications directly. More details on how fees are priced and calculated is in section 4 fees.

Because minting new tokens effectively tax Token Holders (mostly on Users and Developers who are not actively Validators), it is generally preferred to minimize coinbase. However, too small a coinbase combined with insufficient fees can result in reducing interest for Validators to provide compute and capital required for security.

Bitcoin is an example, where the initial design has the coinbase halving every 4 years, with the goal to ultimately sustain the network only with fees. But due to current incentives in longest chain networks where transaction rewards go only to the block producer - there is well-known (Carlsten et al. [2]) issue with instability when the coinbase reward approaches 0.

Thus we suggest a system that sets a ceiling for the maximum coinbase and dynamically decreases the coinbase depending on the amount of total fees in the system. This ensures a minimum epoch reward, and with growth in usage, reduces inflation.

To calculate the actual coinbase reward, we first calculate the maximum inflation per epoch:

$$maxCoinbase = totalSupply_t \times (\sqrt[numEpochsPerYear]{1 + maxInflation} - 1) \quad (2)$$

Where $totalSupply_t = initialSupply + \sum_{i=0}^{t-1} coinbaseReward_i$, meaning it is the total number of tokens in the system at a given epoch t.

Given $maxCoinbase$, we can calculate $coinbaseReward$ for given epoch:

$$coinbaseReward_t = \begin{cases} 0 & epochFees_t \geq maxConibase \\ maxCoinbase - epochFees_t & otherwise \end{cases} \quad (3)$$

Which means, that if the total fees for a given epoch are greater than the maximum coinbase, the fees themselves provide sufficient incentive and the actual coinbase for that epoch can be zero. Otherwise, total fees will decrease inflation by the corresponding amount.

4 Transaction and Storage Fees

Each transaction has a few different components that make up its cost: the cost for receiving and transmitting the transaction (bandwidth), the cost for processing (especially if this is a complicated state transition / smart contract) (CPU) and the cost for state storage (for keeping the information going forward).

Because computation and bandwidth can be charged simultaneously on a per-transaction basis they can be combined into one scalar, usually known as a transaction fee, denoted as $txFee_{index}$.

On the other hand, state persists between transactions and we require all validators that are responsible for a given shard to have it available. This dynamic rents itself naturally to a 'rent model', where accounts are charged to store data for each unit of time. We describe exact mechanics of $stateRent$ in the section 4.3.

In section 3 we used $epochFee_t$, which combined all of the rewards that will be awarded to validators at the end of the epoch. This value is a combination of all state rent fees and $(1 - developerPct)$ transaction fees during the epoch:

$$epochFee_t = \sum_{i=firstBlock_t}^{lastBlock_t} (1 - developerPct) \times txFee_i + stateFee_i \quad (4)$$

4.1 Pricing Computation and Bandwidth

As described above, computation and bandwidth are combined into one scalar, usually referred to as "gas" and defines how many resources are used per transaction.

The exact relationship between 1 CPU instruction and 1 byte of bandwidth is left to implementation details, but the general idea is that total "gas" of given transaction can be computed as:

$$gas_{tx} = numberOfCPUInstructions(tx) + \alpha \times SizeOf(tx) \tag{5}$$

Where α is the relative relation between a unit of computation and a unit of bandwidth. Usually this value is constant (and can be adjusted via governance if it changes drastically). And $SizeOf(tx)$ is the size of the transaction in bytes sent over the wire (i.e. bandwidth).

Each transaction must specify the amount of gas it needs as part of the transaction data. This allows block producers to roughly estimate the amount of gas that will be used when the block will be executed before they actually do it (e.g. as they first must produce a block and only then will execute it).

This amount can be an over-estimation of expected gas usage because the unused amount will be returned to the payer of the transaction after all transactions are completed (all receipts finished across all shards). If a transaction doesn't attach enough gas to execute a required function, the transaction will terminate early and fail, but still charge for spent gas.

Given the gas amount that is specified in a particular transaction, we use a system-wide $gasFee$ variable to calculate the fee in NEAR tokens. That variable defines the current price of 1 unit of gas in the native token.

There are a few considerations about this variable:

- Different shards may have different transaction loads but because of the interaction between shards it would be extremely developer unfriendly if prices were different in different shards. For example, a transaction that touches 3-4 shards would need to pay different and unpredictable fee for each shard.

- Dynamic sharding (defined in 2.1) provides us with a re-balancing technique that maintains a roughly even load across shards. Despite this, re-balancing is not an immediate process and thus there might be some shards that experience congestion in the short term.

- It is good for developers and users to have a low $gasFee$ but it is even more important that it is predictable. One of the main concerns voiced by users of similar systems has been high unpredictability of the price of gas.

- There are two things observed from the blockchain for each block $index$:
  - $gasLimit_{index}$, the maximum amount of gas that is allowed in each shard at that index
  - $gasUsed_{index,shard}$, the amount of gas actually used in each shard at that index

- To facilitate predictable gas pricing, we define a "price slide" $gasFee = gasFee \times (1 + (\frac{gasUsed}{gasLimit} - \frac{1}{2}) \times adjFee)$. Where $adjFee$ is by how much the $gasFee$ can change after each block.

This is similar to the pricing model in Buterin [1] with a difference that instead of defining $minFee$ + tip, we only define $gasFee$ as a standard price.

Because the transaction fees are distributed among all of the validators, who bear the cost of maintaining this network and

Another possible attack is when validators accumulate a large number of transaction to flood blocks with them and raise the $gasFee$. This problem is addressable by adding a strict expiration (TTL) on transactions. This would make collecting transactions during a prolonged period of time not relevant.

There is one consideration that $gasFee$ can go below the minimum amount of marginal cost of accepting a transaction and propagating it to other validators (e.g. bandwidth costs). Validator are motivated to still accept transactions to actually increase the fee back as well as to maintain the network operational (which long term benefits them as well as token holders of the system) and decrease their staked capital inflation. To prevent that, we define $minGasPrice$, which is the floor for gas price. This also helps prevent rounding errors when price is calculated.

It is important to note that this price is universal across all shards. This both simplifies the usage of the network, because now when a transaction that touches multiple shards is issued, the price of it is known ahead of time and can be easily calculated.

The negative effect of such a decision is if one (or a few) shards are at capacity and the rest of the shards are empty - gas prices won't go up. Nevertheless, we believe it is better to rely on dynamic resharding to balance shards out after a period of time to alleviate this imbalance rather than to introduce a more complex rule for adjusting gas price to account for this case.

4.2 Computation limits

As the network evolves we expect that the amount of computation that the network can process will change as well. We generally target lower-end servers and desktops as nodes in the system, but we acknowledge that over time resources available to validators will grow. E.g. what was a high-end computer a few years ago is cheap piece of hardware now. Additionally we expect that software will continue to improve and the way we calculate the amount of resources spent on a specific task may take less real (wall) time even on the same hardware.

To address this concern, we suggest that at every block producers vote on $gasLimit$ by providing a new value within $\pm 0.1\%$ if the previous block has $gasUsed > 0.9 * gasLimit$.

For example if block $index$ has $gasLimit$ of $50,000$ validator can provide any value within $[49,950; 50,050]$ range.

To determine on the node if the limit should be increased or decreased, we should measure the time that the previous block took to validate. If it took

longer then 90% of expected block time, a limit should be reduced, otherwise if it took less time than expected - limit can be increased.

This will allow for the system to dynamically find a limit that $> 50\%+1$ of block producers can maintain.

4.3 Pricing state storage

Because storage is a fundamentally difference resource from computation and bandwidth (whereas those costs are one-time burdens on validators and other nodes) state space must be stored going forward across all the nodes.

Because of this, storage is usually a mispriced resource. If it is paid only once at the inclusion of the transaction - the reward goes to the block producer but requires all other nodes going forward to store information. It's especially true in the case of Bitcoin, where it's expected that every node maintains full UTXO history but only the miner that included it receives the reward.

To solve this problem, we follow Buterin [1] with the suggestion of state rent. For each block time each account is charged $StoragePrice \times SizeOf(account)$ tokens, where $SizeOf(account)$ is size of the account in bytes.

If account's balance goes below $minBalance = pokeThreshold \times storagePrice \times SizeOf(account)$ anyone can send a special transaction that clears state from this account. As a reward the steward gets some $pokeReward$ of the remaining balance on the account. Next few rules also apply:

- If a transaction would bring the balance below $minBalance$, by either moving money or increasing size of the account, this transaction considered failed.
- If this account is staking, e.g. has $staked > 0$, we can not just remove the account. Instead for staking accounts we require $minBalance$ to be $4 \times epochLength \times storagePrice \times SizeOf(account)$. If at the epoch boundary validator has $balance < minBalance$, their proposal or rollover will not be accepted.

Instead of updating balance every step, which is impractical, instead we update the accounts only when they are already changed by some transaction:

- Each account has a $StoragePaidAt$ field.
- Current balance is then calculated $balance - StoragePrice \times SizeOf(account) \times (curBlock - StoragePaidAt)$.
- When account is modified, we recompute the size of the state and update $balance$ given formula above, setting $StoragePaidAt$ at the current block.

Even though cost on storage may change over time, this is a slow process and can be addressed by a network participants decision (through a governance process).

On the other hand, there is volatility in the price between a native token and underlying price that validators pay (and users/developers expect).

In this work, we set $StoragePrice$ as constant and postpone exploration to address volatility to future work.

Worth mentioning also a way to hibernate account by collapsing state of the account to just a Merkle root of the state tree. Then if an account needs to be restored, the original data can be presented. Merkle root is enough to verify that information match and restore data back in its place. At the moment we will not be adding this functionality but expect this to be added soon after release.

5 Validators

Validators play a core function in any ledger by collecting transactions, ordering them, computing new state (by running smart contracts) and provide data to other participants in the system.

In Proof-of-Stake model, security and Sybil resistance of the system is provided via "staking" mechanism, thus requiring validators be Token Holders, maintaining a balance on their account.

Additionally because validators are selected for a period time, they are required to be online. In this design we require just to be online over some threshold $onlineThreshold$.

5.1 Validator Selection

Validator selection is done via an auction mechanism. To become a validator, the node must send a signed transaction which contains information about the amount they want to stake and a new public key that blocks will be signed with. This key can be separate from the keys used to access the user's account. This allows the decoupling of custody of assets from validation and enables other use cases, for example delegation described in section 5.5.

The inclusion of these staking transactions in a block is disincentived for the currently selected group of validators because it effectively increases their competition. We propose to add additional incentive for current validators to include this transaction - new staker defines $inclusionFee\%$ as part of their transaction, which defines how much of their next day's reward will be awarded to block producer that included it.

Staking transactions are collected during the $epochLength$ number of blocks. At the end of the epoch $T-1$ everyone on the network run validator selection process (auction) to determine validators for the epoch $T+1$. Because as part of the validator selection process, we also shuffle validators between shards, it is required that validators assignment is known in advance to perform shard state sync. As network grows, state sync will start taking measurable time so we calculate validators an epoch before.

Given collected set of proposals and validators already computed for epoch T, the next procedure is done.

- Convert validators for epoch T into proposals as well, with their current stake as the bid. If there is competing proposal from the same validator - take it's value.
- Remove proposals of all the validators that were online less than $onlineThreshold$ % or explicitly decided to stop validate.
- Find the $seatPrice$ threshold, such that sum of all stake above $seatPrice$ is greater than number of seats:

$$seatPrice = \arg\max_{x \in \mathbb{N}} \left(\sum_{v \in V} \lfloor \frac{stake_v}{x} \rfloor \geq numSeats \right) \tag{6}$$

- For each validator who has $stake_v \geq seatPrice$, assign them $\lfloor \frac{stake_v}{seatPrice} \rfloor$ seats and randomly shuffle this set.
- If the resulting array has more than $numSeats$, discard the rest.

The resulting ordered array of validator assignments, is then used to split validators into two groups: block/chunk producers and "hidden" validators.

5.2 Reward

To compute individual reward per validator, first need to compute total reward of all validators per epoch, as percentage of $epochReward$:

$$totalValidatorReward_t = (1 - protocolPct) \times (coinbaseReward_t + epochFees_t) \tag{7}$$

The $totalValidatorReward$ is then split between the individual validator in the same proportions if they were a block producer or a "hidden" validator. This also means there is no difference in which shards was this node participating (for "hidden" validator we may never find out).

The validator reward does get adjusted by the % of their online participation in the epoch and the number of seats they hold. The validator reward for validator v is then:

$$validator_v Reward_t = uptime_t^v \times totalValidatorReward_t \times numSeats_v \tag{8}$$

Where $numSeats$ is number of seats this validator got based on their stake, e.g. $stake_v / seatPrice$. And $uptime_v$ is calculated in the next way:

$$uptime_t^v = \begin{cases} 0 & uptimePct_t^v < onlineThreshold \\ \frac{uptimePct_t^v - onlineThreshold}{1 - onlineThreshold} & otherwise \end{cases} \tag{9}$$

Where $uptimePct_t^v$ is percentage of required blocks that this validator produced or validator signed (i.e. uptime). Which means that if given validator

doesn't produce enough blocks in the epoch t, they will not receive any reward and will be removed from the validator pool for the upcoming epoch $t + 2$.

Rewards are automatically re-staked, by adding for any given validator the $validator_v Reward_t$ to their $stake_v$ balance for upcoming epoch.

5.3 Stake slashing

Stake provides security for two reasons:

- Sybil attack resistance to prevent a single entity from taking over all the spots in the validator pool and controlling block production.
- Preventing misbehavior because it causes a certain amount of staked tokens to be slashed (re-distributed among nodes which reported the misbehavior).

We define misbehavior as actions that can be proven cryptographically:

- Double signing a block at the same height.
- Signing a block with an invalid post state root (i.e. invalid state transition)

Note that we are not slashing for a validator that went offline. Validators will be kicked out automatically at the end of the epoch from the rollover pool for the next validator selection auction if they have participated in block production/validation less than $onlineThreshold$.

The incentive for a validator to stay online above $onlineThreshold$ is defined as an opportunity cost of losing their spot in the validator pool, as it will result in losing rewards for at least 2 epochs. Above the $onlineThreshold$ percentage, the incentive for validator is mostly to receive more reward (as detailed in equation 8 as validator misses more blocks - their rewards reduces).

5.4 Stake withdrawal

At any point, validator can issue a transaction to remove themselves from validator pool (same works for changing the amount of stake). This transaction gets recorded on the blockchain. At the $EPOCH_T$ when the new validator selection auction is been performed, we account for this transaction as a proposal for 0 tokens.

After the auction is done, the validator who submitted this 0 token bid will not be in the validators for $EPOCH_{T+2}$. The stake will be returned to their account at $EPOCH_{T+3}$.

As mentioned in section 5.3, if a validator is a block producer and didn't produce enough blocks, or as "hidden" validator didn't provide enough confirmation signatures, its stake will be automatically withdrawn and the validator will be removed from the next auction.

5.5 Delegation

Participating in validation directly benefits token holders both by providing those validators with rewards and indirectly by attracting more participants to join the network because it has greater economic security. Not all capital holders, though, have the ability or desire to run validators.

Instead of defining a specific protocol for delegation, the basic tools provided by the smart contract platform allow for others to create staking smart contracts. For example, a validator can create a smart contract that defines rules of capital allocation and reward distribution alongside participating in staking. This could be used to allow other token holders to delegate their stake to the contract creator.

This same pattern has a wide range of potential uses, including so-called "derivative stake" (see for more details [3]).

6 Developer Business Models

There are few ways applications and their developers can have business models on the blockchain:

1. Users pay a transaction fee and there is an extra that developers add to it.
2. Users buying some items (such as NFTs or token passes).
3. Users paying a flat fee via subscription or another type of charging model.

NEAR already makes it easier for 2 and 3rd option via AccessKeys and different types of interactions between application and user.

On the other hand, in the case of charging a fee on top of the transaction, developers might end up in the situation where their contract is forked with the removed fee. To balance this and the requirement for applications to pay storage rent on the data, we add "developer fee" - a portion of transaction fee does directly to the application balance.

$$developerReward_{index}^{account} = developerPct \times txFee_{index}^{account} \quad (10)$$

Where $txFee_{index}^{account}$ is total amount of gas fees that were used by given *account* at block *index*. The *developerReward* are allocated by per block per account, as they can be efficiently done every time the transaction or receipts is being processed by the contract.

There are different ways this money can be allocated from there and it's up to a developer of the application to decide how to distribute it while maintaining an application.

With this reward program, we also align incentives of developers to attract more users on the platform who will transact more.

7 Protocol Treasury

To fund continues development of the protocol and ecosystem we allocate $protocolPct$ percent of the epoch rewards from section 3 to the designated account. The specifics of governance and management of this account are outside of the scope of this paper.

$$protocolReward_t = protocolPct \times (coinbaseReward_t + epochFee_t) \quad (11)$$

8 Future Work

This design provides general guidance to enable a balance between validator incentives versus developers and users requirements to have stable and predictable pricing of the utility usage. We expect there will need to be more research in the area of aligning incentives between various competing parties to provide a balanced system that is at the same time not subject to be gamed.

One of the directions of future work we suggest is an investigation of using algorithmic stable tokens for pricing resources, using the stake as collateral to provide this stable token a collateral debt position.

References

[1] Vitalik Buterin. Blockchain resource pricing. https://github.com/ethereum/research/blob/master/papers/pricing/ethpricing.pdf, 2019.

[2] Miles Carlsten, Harry Kalodner, S. Matthew Weinberg, and Arvind Narayanan. On the instability of bitcoin without the block reward. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 154–167, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978408. URL http://doi.acm.org/10.1145/2976749.2978408.

[3] Illia Polosukhin. Staking and delegation via smart contract. https://research.nearprotocol.com/t/staking-and-delegation-via-smart-contract/43, 2019.

[4] Alex Skidanov and Illia Polosukhin. Nightshade: Near protocol sharding design. https://nearprotocol.com/downloads/Nightshade.pdf, 2019.

A Constants

For reference, we provide Table 1 as an example values of constants that NEAR Protocol is planning to use while using economics design described in this paper.

Name	Value
$initialSupply$	1,000,000,000 NEAR
$maxInflation$	0.05
$blockTime$	1 second
$numEpochsPerYear$	730
$onlineThreshold$	90%
$pokeTheshold$	500
$storagePrice$	7e−15 NEAR per byte per block
$protocolPct$	10%
$developerPct$	30%
$adjFee$	0.001

Table 1: Important constants in the system (subject to change)

	Compute (100ms/128MB)	Bandwidth	Storage
AWS EC2	$0.000000144 (1)	$0.09 per GB	-
AWS Lambda	$0.000000208 (2)	$0.09 per GB	-
AWS S3	-	0.02−0.09 per GB	$0.02 / GB / month
Heroku	$0.000000964 (3)	-	$0.7815 GB / month (4)
Ethereum (volatile)	$0.1 (5)	...	-
Ethereum proposed state rent	-	-	∼$1342 / GB / month (6)
EOS (volatile)	$0.00016 (7)	$0.26 per GB (8)	∼$888 / GB / month (9)

Table 2: Price comparison between different cloud and blockchain compute providers

B Pricing comparison

Table 2 compares approximate prices (as of April 2019) of different computational platforms. Next assumptions were used for the calculations:

1. t3.micro EC2 with 1Gi RAM and 2 CPUs - $0.0104 per hour $\implies$ $0.000000288 per 100ms $\implies$ run 2 in parallel
2. AWS and specifically AWS Lambda pricing is also extremely complicated: https://medium.com/@zackbloom/serverless-pricing-and-costs-aws-lambda-and-lambda-edge-
3. One simple dyno $25 / month, 100ms in month $\implies$ 25,920,000 requests.
4. Standard 0 - 64GB storage for $50 / month $\implies$ $0.7815 / GB / month.
5. 500ETH per day in tx fees, $150 for ETH, 700k tx per day. 3% of total reward (as of 2019/04/11), not really accounting for 100ms; probably going to cost more.
6. [1] suggests flat price for storage at 1e-7 ETH / byte / year $\implies$ $1342 / GB / month

7. Reserve \$0.013 ms/Day. Loaning \$1.3 for 1 day at 4.5

8. Reserve \$0.002 USD/KiB/Day $\implies$ \$2097 GB/day. Loan for 1 day at 4.5% APR \$0.26

9. Buy price \$0.226 / KiB $\implies$ \$236,978 / GB. Loaning for 30 days at 4.5% APR \$888.67